社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－SE－162 (4)
2008／11／19

適合率を重視したレイヤアーキテクチャ向けソフトウェアクラスタリング手法

加賀洋渡[†1] 新田直也[†2]

実規模ソフトウェアの保守, 再利用においてアーキテクチャの理解は必要不可欠である. しかし, 現実のプロジェクトではアーキテクチャに関する情報は文書化されていないか, 文書化されていても陳腐化している場合がほとんどであり, 保守, 再利用作業において多大なコストを発生している. そのため, ソースコードからアーキテクチャの情報を抽出するソフトウェアクラスタリング技術が広く研究されているが, 多くのソフトウェアクラスタリング手法では, ソフトウェア部品間の利用関係の向きに関する情報が捨象されているため, レイヤアーキテクチャを持つ大規模なソフトウェアのクラスタリングには適さない. そこで我々の研究グループは, 利用関係の向きに着目したクラスタリング手法の研究に取り組んでおり, 本稿では, 文献[15] で提案したクラスタリング手法およびクラスタリングツール SCALAR に対して, 性能向上を目的とした拡張を行った. その結果, 従来手法の高い適合率と実用的な実行速度をほぼ維持したまま再現率を向上させることに成功した.

A Precision-focused Software Clustering Method for Layer Architecture

HITOTO KAGA and NAOYA NITTA

Architecture understanding is crucial for large scale software maintenance and reuse. However, in an actual project architectural documents are often obsolete or rather missing, and a maintainer often have to make a great effort to extract implicit architectural information from the source code. To address the problem, many works have been done in the field of software clustering, but most of the works are not sufficient for clustering large scale software which has a layer architecture because they omit the information of the directions of the use relation. Therefore, we study software clustering methods which are aware of the directions of the use relation, and in this paper, we extend a software clustering method and a clustering tool SCALAR presented in our previous work[15] to improve its performances. As a result, we can improve the performance of recall with preserving high performance of precision and efficiency.

1. はじめに

実規模ソフトウェアの保守, 再利用においてアーキテクチャの理解は必要不可欠である. アーキテクチャを理解する上では, 適切に保守されたアーキテクチャ文書が非常に有用であるが, 現実のプロジェクトではその種の文書が存在しないか, 存在しても陳腐化している場合が多い. その結果, 開発者は保守, 再利用時にソースコードからアーキテクチャに関する情報を取り出さなければならず, そのことが大きな負担となっている.

そのため, ソースコードからアーキテクチャに関する情報を抽出するリバースエンジニアリング技術が広く研究されてきたが (文献[1),5),6),8)–13)]), 本研究ではそのうち, ソフトウェア全体の構造を理解可能な小さなサブシステムに分解する**ソフトウェアクラスタリング** (文献[8)–10)]) と呼ばれる技術に注目する.

従来のソフトウェアクラスタリングでは, 対象となるシステムは比較的小規模なものが多かった. また一般にソフトウェアクラスタリングでは, モジュールやクラスなどのソフトウェア部品間の利用関係を元に解析が行われるが, 利用関係の向きが考慮されていないものがほとんどであった.

そこで我々の研究グループは, ソフトウェア部品間の利用関係の向きに着目し, 大規模なシステムにおいてよく見られるレイヤアーキテクチャ[2)] の構造を抽出するソフトウェアクラスタリング手法の研究に取り組

†1 甲南大学 自然科学研究科 情報システム学専攻
†2 甲南大学 知能情報学部

んでいる．文献[15]では，ソフトウェア部品間の利用関係を有向グラフ G で表し，非循環依存関係の原則に基づいて，G から G の極大強連結成分をクラスタとして取り出すクラスタリング手法を提案した．また，当手法をクラスタリングツール SCALAR として実装し，SCALAR を用いて複数の実規模アプリケーションを対象に評価実験を行った．その結果，当手法によるクラスタリングがいずれのアプリケーションにおいても非常に高い適合率を示すという結果を得ると同時に，得られる再現率が低いという問題点も明らかになった．

本稿では文献[15]の手法を拡張し，高適合率を維持しつつ再現率を向上させるクラスタリング手法の確立を目指す．方針としては，文献[15]の手法が出力する各クラスタを縮約して，新たな非循環有向グラフ G' を構成し，G' に対して適合率の高い別のクラスタリング手法を適用することで再現率を段階的に向上させるアプローチを採用する．ここで適用するクラスタリング手法は本稿で新たに提案するもので，下位レイヤが複数の上位レイヤに利用され易いという性質を利用したものである．具体的には，G' の各頂点に対して，その頂点を終点に持つ極大有向パスの総数を求め，それを各頂点の特徴量として，特徴量が近い頂点同士をまとめるクラスタリングを行う．本稿では，このクラスタリングを利用して SCALAR の拡張を行い，複数の実規模アプリケーションを対象に評価実験を行った．その結果，文献[15]のクラスタリングと比較して，適合率を維持しつつ再現率を向上させることに成功した．

本稿で新たに提案したクラスタリングは非循環有向グラフを非循環有向グラフに縮約する一種のグラフ変換とみなせるが，理論的には，非循環性を保存するグラフ変換は無数に存在する．今後，このようなグラフ変換を追加適用することで，再現率をさらに向上させることができると期待される．

2. レイヤ構造抽出問題

最初に本論文で用いる諸概念について定義する．本論文では，ソースコード全体を**モジュール依存グラフ**で抽象化する．モジュール依存グラフは有向グラフ $G=(V,E)$ で，V はモジュール (ソフトウェア部品) を表す頂点の集合，$E \subseteq V \times V$ はモジュール間の依存関係を表す有向辺の集合である．モジュール依存グラフ G が与えられたとき，G の頂点の集合を $V(G)$ で表し，G の有向辺の集合を $E(G)$ で表す．G の 2 つの頂点 $v, v' \in V(G)$ が $e=\langle v, v' \rangle \in E(G)$ を満たすとき v と v' は**隣接する**といい，v および v' を e の**端点**という．G 中の隣接する頂点列 $p=\langle v_0, \ldots, v_k \rangle$ のうち頂点の重複を許さないものを**パス**といい，$v_0=v_k$ かつ $v_0, \ldots, v_{k-1}$ が重複しないものを**サイクル**という．特に頂点列を結ぶ各有向辺の向きが等しいパスおよびサイクルを，それぞれ**有向パス**，**有向サイクル**という．頂点 $v \in V(G)$ の**入次数**とは，v に入ってくる有向辺の数であり $d_{\mathrm{IN}}(v)$ で表す．同様に v の**出次数**とは，v から出ていく有向辺の数であり $d_{\mathrm{OUT}}(v)$ で表す．グラフ G 中の任意の異なる 2 点間にパスが存在するとき，G は**連結**であるという．また，G 中の任意の異なる 2 点間に有向パスが存在するとき，G は**強連結**であるという．頂点の集合 $V' \subseteq V(G)$ に対して**誘導部分グラフ** $G[V']$ とは，V' を頂点集合に持つ有向グラフで G の有向辺のうち両端点が V' に属するものすべてを有向辺として持つグラフである．G の連結な誘導部分グラフを G の**連結成分**といい，強連結な誘導部分グラフを G の**強連結成分**という．G から有向辺 e を取り除き，その両端点を 1 つの頂点にまとめたものを G の e による**縮約**といい G/e で表す．同様に H を G の誘導部分グラフとしたとき，H のすべての有向辺による G の縮約を G/H で表す．

本研究が対象とするソフトウェアクラスタリングを以下に定義する．モジュール依存グラフ G のクラスタとは G の連結な誘導部分グラフ $G[V']$ のことをいう．G のクラスタ分割とは，G のクラスタの組 $\Pi_G=\langle G_1, \ldots, G_k \rangle$ で $1 \leq i,j \leq k, i \neq j$ のとき $V(G_i) \cap V(G_j)=\phi$ かつ

$$\bigcup_{1 \leq i \leq k} V(G_i)=V(G)$$

を満たすものをいう．Π_G のクラスタ構造 G/Π_G を $G/\Pi_G=((G/G_1)/G_2)\ldots/G_k$ で定義する．ただし $V(G/\Pi_G)=\Pi_G$ とする．本研究ではレイヤアーキテクチャの抽出を目的とするため，G/Π_G はレイヤ間の依存関係を表すものと考える．一般にレイヤ間には非循環依存関係の原則が成り立っているため，G/Π_G は有向サイクルを持たないと考えてよい．このような G/Π_G を特に**レイヤ構造**と呼び，このときの Π_G を**レイヤ分割**と呼ぶ．G が与えられたとき，あるレイヤ分割 Π_G を求める問題を**レイヤ構造抽出問題**という．

一般にレイヤ構造抽出問題は 1 つ以上の解を持つ．特に以下の 2 種類の解は重要である．

例 2.1 (**自明な解**) クラスタ分割 $\Pi_G=\langle G \rangle$ は，常にレイヤ構造抽出問題の解となる．この解を自明な解という．自明な解は明らかにクラスタ数を最小にする解である． □

例 2.2 (**極大強連結成分で構成された解**) G のすべての極大強連結成分を $G_1, \ldots, G_k$ とするとき，クラ

スタ分割 $\Pi_G = \langle G_1, \ldots, G_k \rangle$ は, 常にレイヤ構造抽出問題の解となる. この解はクラスタ数を最大にする解である. □

3. 提案手法

提案手法では, Java のソースコードを解析の対象とする. 任意の Java ソースコードが与えられたとき, 抽象型 (クラス, インターフェース) を頂点とみなし, 抽象型の間の継承関係, 実装関係, フィールドによる参照, メソッドシグニチャによる参照などを含むあらゆる依存関係を有向辺とみなしてモジュール依存グラフを構成する. 構成されたモジュール依存グラフに対して, 本稿では, 文献[15] で提案した手法を拡張したクラスタリング手法を用いてレイヤ構造抽出問題を解く.

3.1 提案手法の概要

文献[15] で提案した極大強連結成分をクラスタとして抽出するクラスタリング手法の拡張を行う. 具体的には, モジュール依存グラフ G 中のすべての極大強連結成分を Tarjan アルゴリズムを用いて抽出したのち縮約し, 非循環グラフ G' を構成する. ここまでの処理を Tarjan 部と呼ぶ. 次に, G' におけるすべての頂点 v で, v を終点に持つ極大の有向パスの数 $\lambda(v)$ を求め, $\lambda(v)$ の値が近い頂点同士をまとめることによってソフトウェアクラスタリングを行う. この処理を PathRank 部と呼ぶ. PathRank 部のアルゴリズムを以下に示す.

3.2 アルゴリズム

非循環有向グラフ G' 中の各頂点 v に対して $\lambda(v)$ を求めるアルゴリズム $PathRank(G')$ は以下の通りである. ただし, 以下では $\mu(v)$ を v の未処理入力辺数とする.

(1-1) 全頂点 v に対して, $\lambda(v) := 0$, $\mu(v) := d_{\mathrm{IN}}(v)$ とおく.

(1-2) $d_{\mathrm{IN}}(v) = 0$ を満たす各頂点 v に対して $\lambda(v) := 1$ とおき, $PathRank(v)$ を実行する.

(1-3) 各頂点 v について $\lambda(v)$ が v への極大有向パス数となる.

ただし,$PathRank(v)$ を以下のように定義する.

(2-1) 頂点 v の出力側に隣接している各頂点の v' に対して以下の (2-2)~(2-3) を繰り返す.

(2-2) $\lambda(v') := \lambda(v') + \lambda(v)$, $\mu(v') := \mu(v') - 1$ とする.

(2-3) $\mu(v') = 0$ の場合, $PathRank(v')$ を実行する.

以上で求めた極大有向パス数の値を用いてクラスタに分割する. 分割は, 閾値 Θ を設定し始点と終点の極大有向パス数の比が Θ を超える有向辺をすべて除去することによって行う.

補題 3.1 非循環有向グラフ $G = (V, E)$ が与えられたとき, $PathRank(G)$ の最悪時間計算量は, $O(|V| + |E|)$ となる.

証明. アルゴリズムより明らか. □

4. 評価実験

解析システムは, 統合開発環境 eclipse[3] 上に, 実装されている. 解析システムは以下の 2 つのツールから成る.

- **MDGExtractor**: Java のソースコードを解釈しモジュール依存グラフを生成する.
- SCALAR**(Software Cluster Analyzer for Layer ARchitecture)**: モジュール依存グラフを元にクラスタリングを行う.

MDGExtractor は eclipse のプラグインとして Java により実装されたものである. ソースコード内のクラスやインタフェースの利用関係を抽出する上で必要となる抽象構文木は eclipse が作成したものを利用している. 本稿では上記ツールのうち SCALAR の拡張を行った.

解析システムを用いて, 本節では提案手法がソフトウェアアーキテクチャの理解のために有効なレイヤ構造を抽出するかについて評価する. 解析対象として Radish と ArgoUML 0.19.2[14] , eclipse 3.2.2 のソースコードを用意した. Radish は我々の研究室で開発した対戦型 3D ゲームフレームワークでソースコードは 2866 行である. ArgoUML はオープンソースの UML エディタで, ソースコードは約 10 万行である. eclipse はオープンソースの統合開発環境で, ソースコードは約 140 万行である. まず MDGExtractor を用いて解析対象のソースコードからモジュール依存グラフを抽出し, その際の処理時間を記録する. この際, eclipse のワークスペース上のプロジェクトを入力ソースコードとし, 抽出したモジュール依存グラフはテキスト形式で出力される. 次に, MDGExtractor が生成したモジュール依存グラフを SCALAR で読み込み解析する.

表 1 は Radish, ArgoUML と eclipse それぞれの頂点数, 辺数, MDGExtractor および SCALAR 内の Tarjan 部と PathRank 部の実行時間を表している.

表 2 は Tarjan 部のみを用いた SCALAR の解析結果である. これは文献[15] の提案手法 1 に相当する. 抽出されたクラスタ C のうちサイズの大きいものに対して, C のサイズ, C に最も一致するレイヤ L, L に対する C の適合率および再現率を示す. 適合率は, クラスタ C があるレイヤ L に対応するとき, C に含まれ

る要素がどれくらいの割合で L に含まれているかを示す．再現率は，L に含まれる要素がどれくらいの割合で C に含まれているかを示す．

同様に表 3 は PathRank 部を用いて解析を行った結果である．ただし，表 2 では C および L は抽象型を単位としているのに対し，表 3 では Tarjan 部によって抽出されたクラスタを単位としている．いずれのクラスタにおいても適合率は 100%と高い値を示していることがわかる．

表 4 は Radish の解析における平均適合率，平均再現率，平均クラスタサイズを Tarjan 部のみと提案手法全体 (Tarjan 部+PathRank 部) で比較したものである．

ArgoUML の Tarjan 部による解析結果および PathRank 部による解析結果をそれぞれ表 5,6 に，同様に eclipse の解析結果を表 7,8 に示す．PathRank 部の解析においてはいずれも閾値 $\Theta = 1.08$ とした．

表 2 Tarjan による Radish の解析結果

クラスタサイズ	適合率	再現率	該当レイヤ
6	100.0%	17.1%	main
5	100.0%	45.5%	model
2	100.0%	40.0%	physics

表 3 PathRank による Radish の解析結果 ($\Theta = 1.08$)

クラスタサイズ	適合率	再現率	該当レイヤ
5	100.0%	14.3%	main
4	100.0%	57.1%	animation
3	100.0%	27.3%	model
2	100.0%	40.0%	physics
2	100.0%	5.7%	main
2	100.0%	5.7%	main

表 4 Tarjan と提案手法の解析結果の比較 (Radish)

	Tarjan	提案手法
平均適合率	100.0%	100.0%
平均再現率	8.7%	12.1%
平均クラスタサイズ	1.22	1.70

表 5 Tarjan による ArgoUML の解析結果

クラスタサイズ	適合率	再現率	該当レイヤ
515	100.0%	41.3%	org.argouml
79	100.0%	28.9%	org.tigris.gef
27	100.0%	12.6%	org.argouml.model ru.novosoft

表 6 PathRank による ArgoUML の解析結果 ($\Theta = 1.08$)

クラスタサイズ	適合率	再現率	該当レイヤ
301	80.4%	24.1%	org.argouml
74	98.6%	34.6%	org.argouml.model ru.novosoft
22	100.0%	1.8%	org.argouml
20	100.0%	7.3%	org.tigris.gef
17	100.0%	1.4%	org.argouml

表 7 Tarjan による eclipse の解析結果

クラスタサイズ	適合率	再現率	該当レイヤ
596	100.0%	20.0%	org.eclipse.ui
455	100.0%	8.75%	org.eclipse.jdt
253	100.0%	4.87%	org.eclipse.jdt
160	100.0%	3.08%	org.eclipse.jdt org.eclipse.jdi
155	100.0%	10.3%	org.eclipse.pde
148	100.0%	12.0%	org.eclipse.team
139	100.0%	15.4%	org.eclipse.core
130	100.0%	8.66%	org.eclipse.pde

表 8 PathRank による eclipse の解析結果 ($\Theta = 1.08$)

クラスタサイズ	適合率	再現率	該当レイヤ
284	90.1%	6.2%	org.eclipse.ui org.eclipse.jface org.eclipse.swt
258	71.9%	5.0%	org.eclipse.jdt org.eclipse.jdi
183	98.4%	3.5%	org.eclipse.jdt
127	66.1%	14.0%	org.eclipse.pde
87	89.3%	23.3%	org.eclipse.ant

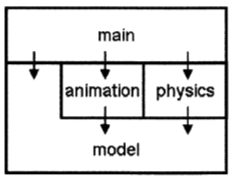

図 1 Radish のアーキテクチャ

5. 評価および考察

Radish に対する解析結果の比較 (表 4) を見る限り，本手法は，従来手法の高い適合率を維持したまま再現率を向上させることに成功しているといえる．また表 1 から，本手法の拡張によって若干の実行速度の低下が認められるものの，全体としてほぼ変わりない実行速度を得られていることがわかる．とりわけ，すべての解析において PathRank 部の処理速度が Tarjan 部の処理速度を上回っていたことは注目に値する．Radish のアーキテクチャを図 1 に示す．Radish は，main, animation, physics, model の 4 つのサブシステムに

表 1 解析システムの実行時間 (Tarjan, PathRank)

	頂点数	辺数	実行時間 (ミリ秒) †		
			MDGExtactor	SCALAR	
				Tarjan	PathRank
Radish	58	814	2,328	78	62
ArgoUML	2228	16601	119,359	18,593	7,470
eclipse	18092	222700	2442,766	8,151,266	1,877,421

† OS:WindowsXP Home Edition, CPU:Intel Celeron 3.33GHz, メモリ:504MB, JVM build 1.6.0.03

表 9 Tarjan と提案手法の解析結果の比較 (ArgoUML, eclipse)

	総クラス数	総クラスタ数		平均クラスタサイズ	
		Tarjan	提案手法	Tarjan	提案手法
ArgoUML	2228	1587	477	1.40	4.67
eclipse	18092	13169	6523	1.37	2.77

よって構成される. これら 4 つのサブシステムのうち, 従来手法では animation サブシステム内のクラスタが 1 つも抽出されていなかったのに対し, 本手法ではサイズ 4 のクラスタが抽出されている.

表 5~8 で示した ArgoUML および Eclipse の解析結果は, Tarjan 部 と PathRank 部で適合率や再現率の計算方法に多少の差があるものの, 高い適合率を維持していることがわかった. ただ一方で, 本手法が適合率の若干の低下を招いていることも明らかになった. PathRank 部によるクラスタリングで特に適合率が低かったクラスタの内部を調べたところ, 上位レイヤ L_1 に, 下位レイヤ L_2 中の L_1 のみに利用されている部品が付随しているという形態のものが多かった. これは本手法特有の性質によるもので, これを本質的に改善することは困難であると考えられる. ただしこのようなクラスタは, 複数レイヤに跨る関連の強い部品群を捉えていると考えることができる. 再現率に関しては, これらの結果だけでは正確な比較ができないが, Tarjan 部 によって抽出できなかったクラスタを PathRank 部 が抽出していることから, 本手法によって真に改善していることがわかる. また, 表 9 において, ArgoUML, eclipse ともに従来手法より提案手法の方が平均クラスタサイズが増大していることからも, 再現率の向上を推測することができる.

なお, ArgoUML および Eclipse の解析結果においては, Tarjan 部と PathRank 部の間で, 抽出された上位クラスタが所属するサブシステムの集合に大きな隔たりは認められなかった. いずれの手法においても, アーキテクチャにおいて重要な役割を果たすサブシステム (文献[4] 参照) が上位に抽出される傾向が認められる.

6. 関連研究

Bunch[9] は疎結合, 高凝集の設計原理に基づいてクラスタリングを行う非常に強力なツールである. しかし, 文献[9] で例題として用いられている対象システムはいずれも規模が小さく, 実用規模のソフトウェアに対して有効であるか否かは不明である. また, この手法ではモジュール依存グラフの有向辺の向きが考慮されていないため, レイヤアーキテクチャなど依存関係の向きが本質的な役割を果たすようなアーキテクチャの抽出に適さないと考えられる.

なお, 文献[8)-10)] 等で指摘されている偏在モジュールが解析結果に与える影響は, 本手法ではアルゴリズムの性質として偏在モジュールが下位レイヤとして分離されるため, 問題とならない.

文献[7),13)] では, モジュール依存グラフの有向辺の向きを積極的に考慮して支配頂点とそれに続く部分グラフを抽出する手法が紹介されているが, 本稿で提案しているクラスタリング手法はこれの一般化に相当する.

我々の研究グループは, 文献[15)] において, 極大強連結成分抽出に基づくクラスタリング手法を提案した. この手法を用いると非常に高い適合率を得れることがわかったが, 再現率においてはよい結果は得られなかった. また同文献で, 再現率の向上を目指してもう一つのクラスタリング手法を考案したが, 適合率において顕著な低下を招いたことと, 実行において膨大な時間を要することから実用性に乏しいものであった. そこで本稿では, 高適合率を維持したまま再現率を向上させる方策を考案し, 文献[15)] の手法の改善に成功した.

7. おわりに

高適合率を維持したまま再現率を向上させることを

目指して，文献[15]のクラスタリング手法および解析ツールSCALARの拡張を行った．また，SCALARを用いて実規模ソフトウェアを対象に評価実験を行った．その結果，適合率において若干の低下が認められるものの，再現率の向上に成功した．また，大規模なソフトウェアに対しても実用的な実行時間でクラスタリングできることがわかった．このことから，本手法を実規模ソフトウェアのアーキテクチャ理解のさらなる支援に用いることができると期待される．本手法は適合率の高い複数のクラスタリング手法を順次適用することで，再現率を段階的に向上させることができるという特長を持つ．今後，適合率の高い別のクラスタリング手法を考案することによって，再現率をさらに向上させることが期待される．

参考文献

1) D. Beyer, A. Noack and C. Lewerents: Simple and efficient relational querying of software structures, In proc. of WCRE'03, 2003.
2) F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal: Pattern-oriented software architecture: a system of patterns, John Wiley & Sons, 1996.
3) Eclipse.org: eclipse, http://www.eclipse.org/, 2007.
4) E. Gamma and K. Beck: Contributing to eclipse: principles, patterns, and plug-ins, Addison Wesley, 2004.
5) R. L. Krikhaar: Reverse architecting approach for complex systems, In proc. of ICSM'97, 1997.
6) J. Knodel, D. Muthig and M. Naab: Understanding software architectures by visualization – an experiment with graphical elements, In proc. of WCRE'06, 2006.
7) S. Li and L. Tahvildari: A service-oriented componentization framework for Java software systems, In proc. of WCRE'06, 2006.
8) J. Luo, L. Zhang and J. Sun: A hierarchical decomposition method for object-oriented systems based on identifying omnipresent clusters, In proc. of ICSM'05, 2005.
9) B. S. Mitchell and S. Mancoridis: On the automatic modularization of software system using the Bunch tool, IEEE trans. on Software Engineering, 32(3), pp.193–208, 2006.
10) H. A. Müller, M. A. Orgun, S. R. Tilley and J. S. UHL: A reverse engineering approach to subsystem structure identification, Journal of Software Maintenance: Research and Practice, 5, pp.181–204, 1993.
11) H. A. Müller, K. Wong and S. R. Tilley: Understanding software sustems using reverse engineering technology, ACFAS'94, 1994.
12) R. W. Schwanke: An intelligent tool for reengineering software modularity, In proc. of ICSE'91, 1991.
13) V. Tzerpos and R. C. Holt: ACDC: An algorithm for comprehension-driven clustering, In proc. of WCRE'00, 2000.
14) Tigris.org: ArgoUML, http://argouml.tigris.org/, 2007.
15) 加賀洋渡, 新田直也: レイヤーアーキテクチャのためのソフトウェアクラスタリング手法, 情報処理学会研究報告, 2008-SE-159(19), 2008.